курс «Формальная спецификация и верификация программ»

Методическое пособие по теме «формализация требований к функции»

Рассматриваются примеры задач, в которых требуется, начиная с неформальной постановки задачи, провести несколько шагов формализации и в результате построить формальные спецификации типов данных и операций над ними.

1. **Пример**. Функция перевода значения температуры по шкале Цельсия в значения по шкале Фаренгейта.

Решение.

value Celsius2Fahrenheit : **Real** -~-> **Real --** как известно самая низкая температура по Цельсию это -273,15 градусов. То есть не любое действительное число может быть аргументом этой функции, следовательно — это частично вычислимая функция, поэтому в описании сигнатуры надо использовать символ "-~->"; второе, аргументом функции может быть только действительное число, которое не меньше чем эта величина - это будет указано в предусловии функции.

```
Celsius2Fahrenheit (c) is (c + 32.0)*5.0/9.0 -- по этой формуле вычисляется
                                             -- значение температуры по шкале Фаренгейта
pre c >= 0.0-273.15
```

2. Пример. Функция поиска минимального значения в списке, которая в качестве результата дает найденное значение.

Решение.

```
value minimum : Int-list -~-> Int   -- функция не определена на пустом списке,
                                    -- поэтому она не total,
                                    -- это ограничение описано в пред-условии
     minimum(nl) as min
     post ~exists m:Int :- (m isin elems nl /\ m > min)
                                    -- результат min корректен,
                                    -- если в списке нет меньшего него числа
     pre nl ~= <..>
```

3. Пример. Функция вставки элемента в «очередь с приоритетом».

Решение.

```
type Data, Queue = (Nat >< Data)-list --
```
элементом очереди является пара (приоритет >< данные)

```
variable queue : Queue   -- переменная queue — это состояние системы,
                         -- функция put зависит от состояния и изменяет его
```

-- пусть при равных приоритетах элементы устанавливаются ближе к концу очереди (б***о***льшие индексы), то есть новые элементы «встают в очередь» со стороны б***о***льших индексов и «обслуживаются» и покидают очередь с головы очереди, позиции 1.

```
value put : Nat >< Data -> write queue Nat        -- результат функции состоит в
```
установке пары «приоритет-данное» в очередь - меняется значение переменной queue, значением самой операции является номер пары в очереди, по которому можно судить - как долго этот элемент будет ждать «обслуживания».

-- функция всюду вычислимая, она определена для пустой очереди, на длину очереди ограничения нет, нет также ограничений на значения приоритета и данных, поэтому предусловие функции тривиальное - **true**, значит его можно вообще не писать.

О постусловии:

- все элементы в очереди, которые находились в ней перед выполнением функции сохранились, при этом
  - элементы с большим или равным приоритетом сохранили свои номера в очереди
  - все элементы с меньшим приоритетом отодвинулись на одну позицию
- новый элемент занял некоторую позицию в очереди

Замечание. Часто при разработке формальных спецификаций становится очевидным, изначальные определения сигнатур функций, типов и даже набора функций некоторого класса можно улучшить в том или ином плане. Такого рода улучшения не обязательно, но желательны. Приведем пример улучшения здесь.

Мы сказали, что дополнительных ограничений на входные данные нет, однако сигнатура функции может быть более информативной:

- Приоритет задается как **Nat**. Можно предположить, что в процессе развития системы приоритет будет и другой природы, например, строка или действительное число. Поэтому лучше ввести новый тип данных, например, Priority, и для сравнения двух значений приоритета ввести функцию ge : Priority >< Priority -> **Bool**.
- Значение-результат функции put является натуральным, но не произвольным натуральным числом - это значение не может быть нулевым, поэтому также вместо неинформативного **Nat** лучше ввести новый тип данных, например, Index = **Nat**.

Таким образом преобразуем декларацию функции put к более информативному виду:

```
type Data, Priority, Queue = (Priority >< Data)-list,
     Index = {| n : Nat :- n > 0 |}
```

```
variable queue : Queue
value ge: Priority >< Priority -> Bool
value put: Priority >< Data -> write queue Index

-- теперь запишем пост-условие
    put (pr, d) as ind
    post
     ind <= len queue` + 1 /\
     len queue` + 1 = len queue /\
     (all k:Index :- k < ind =>
           queue(k) = queue`(k) /\
           ge(let (pe, de) = queue(k) in pe end, pr) ) /\
     queue(ind) = (pr,d) /\
     (all k:Index :- k <= len  queue /\ k > ind =>
           queue(k) = queue`(k-1) /\
           ~ge(let (pe, de) = queue(k) in pe end, pr)  )
```

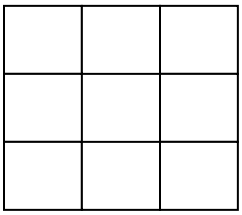
Замечание 1. Похоже, после того, как мы записали семантически важные ограничения, мы должны позаботиться о тотальной вычислимости пред- и постусловий, тогда появляется дополнительные ограничения.

Замечание 2. Можно сформулировать инвариант для Queue – приоритет с ростом индекса не возрастает.

4. Пример. Игра "крестики-нолики". tick-tack-toe

Решение.

Поле для игры — таблица «3 >< 3».



Пусть каждая клетка идентифицируется парой цифр от 1 до 3.
Имеется два игрока, ход первого представляется операцией cross_move, ход второго – tack_move. Состояние таблицы, пусть представлено таблицей table, в каждой клетке которой стоит или крестик – cross, или нолик – tack. Если клетка пустая, то будем считать, что в данный момент этой клетки в таблице (как структуры данных) просто нет.
Имеется функция, которая инициирует новую игру — new_game.

Опишем типы данных и сигнатуры функций:

```
type Ind = {| n : Nat :- n isin {1..3} |} -- координата клетки
                                          -- изменяется от 1 до 3
type CellValue
value cross, tack : CellValue    -- константы cross и tack являются константами sort-типа
axiom cross ~= tack              -- CellValue, важно чтобы они были различными
type Cell = {| c : CellValue :- c=cross \/ c=tack |}
                                 -- тип Cell является подтипом CellValue
variable table : (Ind >< Ind) -m-> Cell := []
      -- начальное значение переменной table это пустое отображение
      -- таков же эффект выполнения функции new_game

      -- инвариант типа Table:
       -- 1. число крестиков равно числу ноликов или превосходит его на 1,
             -- (то есть предполагается, игру всегда начинают крестики)
axiom forall t : Table :-
   let cc = card {cell : (Ind >< Ind) :- cell isin dom t /\ t(cell) = cross},
       ct = card {cell : (Ind >< Ind) :- cell isin dom t /\ t(cell) = tack} in
                 cc = ct \/ cc = ct+1 end,
      -- 2. в таблице не может быть одновременно две выигрышные позиции
      -- для крестиков и для ноликов:
axiom forall t : Table :- ~( gameover(cross,t) \/ gameover(tack,t) )

value
     new_game : Unit -> write table  Unit
     new_game() as () post table = [],

     cross_move : Ind >< Ind -~-> write table  Unit
     cross_move(k, l) as ()
```

```
-- содержимое всех клеток сохранилось
-- и новое значение установлено на соответствующее место:
post dom table = dom table` union {(k, l)}           /\
     (all i, j : Ind :- (i,j) isin dom table`  =>
                        (table(i,j) = table`(i,j)) /\
     table(k,l) = cross

 -- пока нет выигрыша, и параметры (k, l) указывают на пустую ячейку:
 pre  ~gameover() /\ (k,l) ~isin dom table,

 tack_move : Ind >< Ind -~-> write table Unit
 tack_move(k, l) as ()
 post dom table = dom table` union {(k, l)}                  /\
      (all i, j : Ind :- (i,j) isin dom table` =>
                              (table(i, j) = table`(i, j))  /\
      table(k,l) = tack
 pre  ~gameover() /\ (k,l) ~isin dom table,

 gameover: Unit -> read table Bool
 gameover() is (gameover(cross,table) \/ gameover(tack,table)),

 gameover: Cell >< Table-> read table Bool
 gameover(cv, tb) as rc

 -- в таблице уже есть выигрышная комбинация игрока cv
 -- (то есть играющего крестиками или ноликами),
 -- если значением cv заполнена:
              -- одна из диагоналей
              -- одна из горизонталей
              -- одна из вертикалей
 post rc =
           -- две диагонали
       ( all i : Ind :- (i,i) isin dom tb /\ tb(i,i) = cv)      \/
       ( all i : Ind :- (i,4-i) isin dom tb /\ tb(i,4-i) = cv)  \/
         -- горизонтали вертикали
       ( exists i : Ind :-
             ( all j : Ind :- (i, j) isin dom tb /\ tb (i, j) = cv) \/
             ( all j : Ind :- (j, i) isin dom tb /\ tb (j, i) = cv)  )
```